Operating Systems (2INC0)

Concurrency and Atomicity (05)
Introduction

Dr. Geoffrey Nelissen

**Courtesy of Prof. Dr. Johan Lukkien and
Dr. Tanir Ozcelebi**

IRiS
Interconnected
Resource-aware
Intelligent Systems

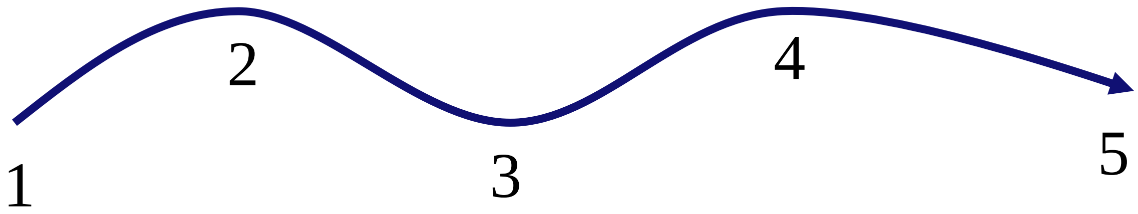
TU/e Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Starting point: the *sequential task*

Discrete sequence of states (e.g., observable in program code):

- needed: indivisible, <u>*atomic*</u> steps/actions between the states
- execution never observed to be half-way an atomic action



Execution: path through state-space

Initially:
StateP = 1

Program:
StateP := 2;
StateP := 3;
StateP := 4;
StateP := 5;

In a task, the finest level of detail w.r.t. progress consists of *atomic actions.*

Atomic? (consider shared variables x, y and atomic memory operations)

- *x := 1*
  - mov *#1, r1*; str *r1, @x*
  - no 'internal' interference point, hence to be regarded as atomic, assuming a correct implementation of interrupt handling (note: actual execution not atomic)

- *x := y*
  - mov *@y, r1*; str *r1, @x*
  - 'internal' interference point: r1 may store an old copy of *y* for a long time while computations with *y* continue.

- *x := x+1 (similar problem)*
  - mov *@x, r1*; inc *r1*; str *r1, @x*

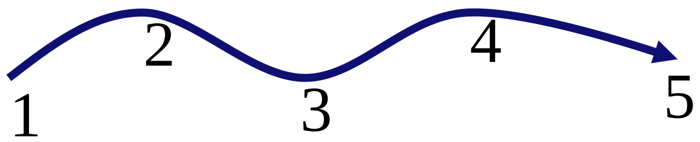
What about z := x; x := z+1 (z private variable)?

→ 2 atomic statements
interference point visible in the program

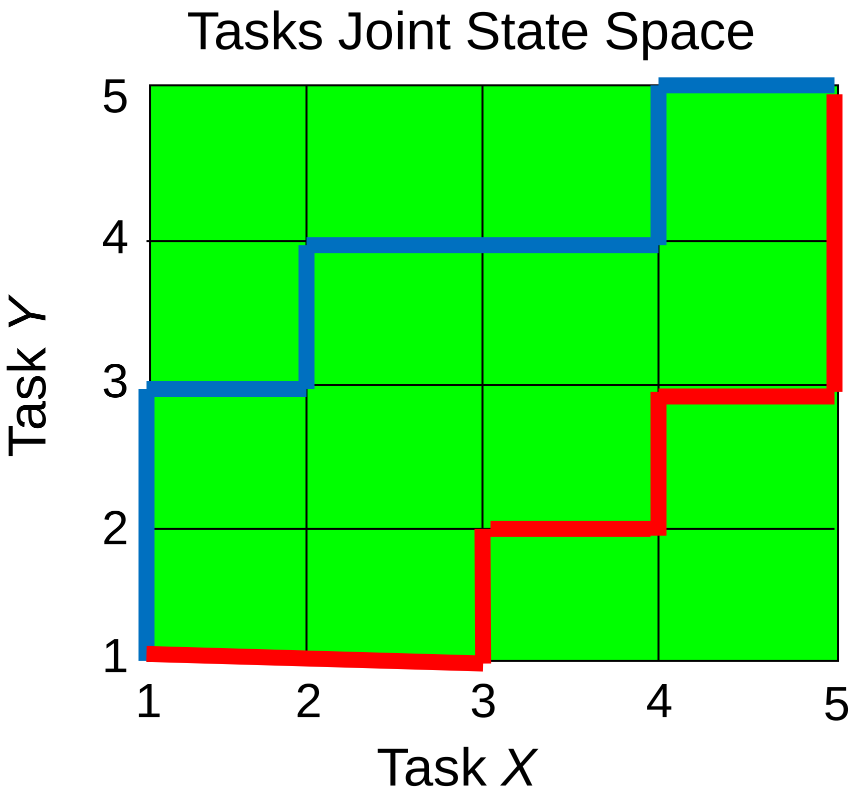
- **Single reference rule**: *a **statement** (expression) in a programming language may be regarded as **atomic if at most one reference to a shared variable occurs** (we ignore here compiler optimizations).*

- **Defined atomicity:** *When we want to regard a non-atomic statement S as atomic, we write* ***< S >*** , e.g. < *x := x+1* >.
  - needs OS and/or hardware support…

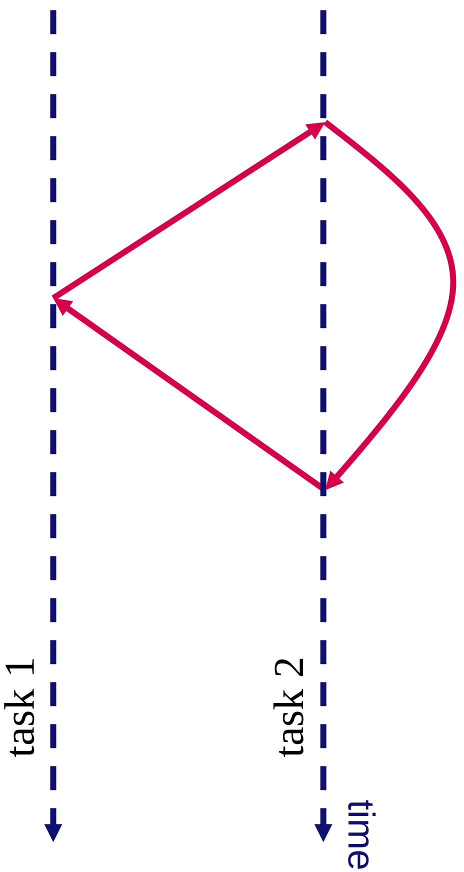
Concurrent execution



- Execution path of a single task (above)
- Concurrent processes take a joint path through the joint state space.
- ***<u>Trace</u>*: sequence of atomic actions**, obtained by **interleaving** the execution of concurrent tasks
  - **maintains** the internal **execution order of the individual tasks**
- **many possible traces** in the joint state space (e.g. <u>blue</u> and <u>red</u> traces)



Traces of concurrent tasks



- **In between any pair of instructions** of one task**,** (part of) **another task** or collection of tasks **can be executed,** including the OS**.**
- **Problem: Old copies of shared variables** can be stored in internal registers or in memory locations.
- Example:
  - initially: *x=1, y=2*
  - program: *x := y || y := x*

mov @x, r; str r, @y

mov @y, r; str r, @x

- What are the possible traces? … the final values?
  - *1) mov @x, r; str r, @y; mov @y, r; str r, @x* → ***(1,1)***
  - *2) mov @y, r; str r, @x; mov @x, r; str r, @y* → ***(2,2)***
  - *…*
  - *6) mov @x, r; mov @y, r; str r, @y; str r, @x* → ***(2,1)***

  - **Exercise:** Is the result *(1,2)* possible?
  - **Exercise:** Assuming all traces are equally likely, what is the probability of the result being *(1,1)*? How about *(2,1)*?

More on traces

- **Trace = sequence of (atomic) <u>actions</u>**;
  It represents the possible steps of a program execution
  - **<u>actions</u>** = **assignments** or **tests**
- **<u>Possible</u> trace = trace in which all the tests yield true**
  - being possible depends on the initial program state
- Traces can be **finite or infinite**, and a program text has many traces

Example

- Consider two concurrent programs:

```
x := false;                 ||     y := E;
while ( not x ) {                  x := true;
    skip;
}
```

Example traces of the left program **(mind the notation)**:

- *(x:=false)(x)*
  (x:=false)(¬x)(skip)(x)
  (x:=false)(¬x)(skip)(¬x)(skip)(x)
  ....
  - note: *(x)* and *(¬x)* denote tests
- **For this program in isolation** (without the right task):
  Only "the infinite trace" is possible, because *(x)* never yields *true.*

More on traces (2)

- The traces of a concurrent program are obtained by interleaving traces of all concurrent parts. Example: interleave *(x:=false)(x)* with *(y:=E)(x:=true).*
  - *(y:=E)(x:=true)(x:=false)(x)*
  - ***(y:=E)(x:=false)(x:=true)(x)* → *possible***
  - *(y:=E)(x:=false)(x)(x:=true)*
  - ***(x:=false)(y:=E)(x:=true)(x)* → *possible***
  - *(x:=false)(y:=E)(x)(x:=true)*
  - *(x:=false)(x)(y:=E)(x:=true)*

REMEMBER:
A trace is a possible one if all its tests yield true

- Note: **Finite traces are now possible traces.**
  - For example, the **two traces in bold** are now possible, while *(x:=false)(x)* was not a possible trace for single program in isolation

Summary

- **Shared variables:** accessible to several tasks (processes/threads)
- **Private variables:** accessible only to a single task
- **Atomic action:** finest grain of detail, indivisible
  - typically, assignments and tests in a program
  - **sufficient at program level: single reference to shared variable in statement**
    - ignoring possible optimization and reordering by compiler/processor
- **Concurrent execution:** interleaving of atomic actions
- **Interference:** disturbing others' assumptions about the state
  - usually, caused by "bad" interleavings
  - particularly with shared variables
- **Race condition:** situation in which correctness depends on the execution order of concurrent activities ("bad interference")